

# MODAL MAP 474D

## Tutoriel Python

Florent Benaych-Georges

Département de Mathématiques Appliquées, École Polytechnique, Palaiseau

2016

Programmes présentés en amphithéâtre disponibles sur  
[http://www.cmapx.polytechnique.fr/~benaych/Slides\\_Python\\_MODAL/](http://www.cmapx.polytechnique.fr/~benaych/Slides_Python_MODAL/)

- 1 Introduction
- 2 Types et conteneurs simples
- 3 Syntaxe : boucles, tests et fonctions
- 4 Import de fonctions, utilisation de bibliothèques
- 5 Programmation matricielle avec `numpy`
- 6 Python pour les probabilités
- 7 Affichage graphique
- 8 Approche empirique en probabilités via Python : exemples
- 9 Lecture et écriture dans un fichier externe
- 10 Python pour l'analyse

- Première version : 1991 (Guido van Rossum, Pays-Bas).  
Versions 2.7 et 3 : quasiment **identiques** pour ce qui nous concerne.
- **Langage de script** (haut niveau, loin du langage machine)
- **Interprété** (*bytecode* compilé en fait)
- **Multi-usage** (**calcul scientifique**, web, interface graphique,...)
- Syntaxe très **lisible** et **épurée**
- **Langage Open Source**, nombreux outils standard disponibles (*batteries included*), plusieurs milliers de packages disponibles dans tous les domaines
- Nombreux interpréteurs et environnements de développement disponibles (ex : **Spyder**, dans la distribution **Anaconda**)
- Communauté d'utilisateurs très active (**StackOverflow.com**)

# Langage interprété vs compilé

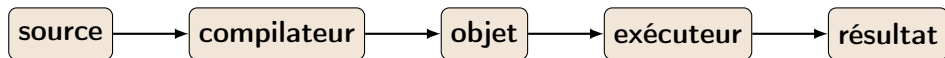
- Langages **interprétés** : **Python**, Matlab, Scilab, Octave, R, ...
- Langages **compilés** : C, C++, Java, ...

Vitesse d'exécution : **interprété** < **compilé**

*Pourquoi, dans ce cas, considérer **Python** pour du calcul scientifique ?*

- *Temps d'implémentation vs temps d'exécution* : **Python**, langage lisible et épuré  $\implies$  développement et maintenance rapides
- *Exécution en **Python*** : **rapide** si les passages critiques sont exécutés avec un langage compilé : de nombreuses fonctions sont compilées et le code est interfaçable avec C/C++/FORTRAN

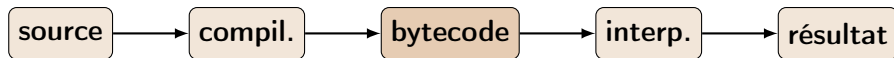
## En fait : *précompilation* du code beamer



: Chaîne de compilation (C, C++, Java,...)



: Chaîne d'interprétation



: Pour exécuter un programme, *Python* charge le fichier source *.py* en mémoire vive, en fait l'analyse (lexicale, syntaxique et sémantique), produit le *bytecode* et enfin l'exécute.

# Fonctionnement

1 Ouverture de l'environnement de développement (ex : Spyder).  
*Non indispensable mais conseillé.*

2 Au choix :

- Commande en ligne
- **Ecriture d'un programme → exécution du programme**
- **Ecriture d'une fonction → chargement de la fonction → appel à la fonction** (d)

Ecriture des programmes/fonctions dans un éditeur de texte quelconque.

Console **IPython** ou **Python** (**Python** seulement pour les programmes dynamiques).

Tutoriel dans la console : **? nom\_fonction** (d)

# Types simples (1) : bool (booléens)

- Deux valeurs possibles : **False**, **True**

```
type(False) # bool  
x=(1<2) # True  
type(x) # bool
```

- Opérateurs de comparaison : **==**, **!=**, **>**, **>=**, **<**, **<=**

```
2 > 8 # False  
2 <= 8 < 15 # True
```

- Opérateurs logiques : **not**, **or**, **and**

```
(3 == 3) or (9 > 24) # True  
(9 > 24) and (3 == 3) # False
```

## Types simples (2) : int, long, float, complex

```
type (2**40) # int
type (2**100) # long
#long: uniquement limite par la RAM
type (3.6) # float
type (3+2*1j) # complex
20+3 # 23
20*3 # 60
(3*1j)**2 # (-9+0j)
20 ** 3 # 8000 (puissance)
20 / 3 # 6 (division entiere en Python 2)
20 // 3 # 2 (division entiere en Python 3)
20./3 # 6.666666666666667
20 % 3 # 2 (modulo)
(2+3j) + (4-7j) # (6-4j)
(9+5j).real # 9.0
(9+5j).imag # 5.0
abs (3+4j) # 5.0 : module
```



## Types/conteneurs simples (3) : str (chaîne de caractère)

```
type('abc') # str
c1 = "L'eau vive"
c2 = ' est "froide" !'
c1+c2 # (concatenation)
c1*2 # "L'eau viveL'eau vive" (repetition)
c1[2] # 'e'
c1[-2] # 'v'
c1[2:5] # 'eau'
len("abc") # 3 (longueur)
"abcde".split("c") # ['ab', 'de'] (scinde)
'a-ha'.replace('a', 'o') # 'o-ho'
'-' .join(['ci', 'joint']) # 'ci-joint'
'abracadabra'.count('bra') # 2
'PETIT'.lower() # 'petit'
'grand'.upper() # 'GRAND'
```

# Conteneurs simples : list

**list** : collection hétérogène, ordonnée et modifiable d'éléments séparés par des virgules, et entourée de crochets.

```
my_list=[4,7,3.7,'E',5,7]
type(my_list) # list
len(my_list) # 6
my_list[0] # 4
range(5) # [0,1,2,3,4]
range(3,6) # [3,4,5]
range(2, 9, 2) # [2, 4, 6, 8]
my_list[1:3] # [7, 3.7]
[0,1] + [2,4] # [0,1,2,4] (concatenation)
```

## Conteneur list : quelques méthodes

```
nombres = [17, 38, 10, 25, 72]
nombres.sort()
nombres # [10, 17, 25, 38, 72]
nombres.append(12)
nombres # [10, 17, 25, 38, 72, 12]
nombres.reverse()
nombres # [12, 72, 38, 25, 17, 10]
nombres.remove(38)
nombres # [12, 72, 25, 17, 10]
nombres.index(17) # 3
nombres[0] = 11
nombres[1:3] = [14, 17, 2]
nombres # [11, 14, 17, 2, 17, 10]
nombres.count(17) # 2
mean(nombres) # moyenne
std(nombres) # ecart-type
```

Par défaut, les objets en Python ne sont pas copiés.

```
x = [4, 2, 10, 9, "toto"]
y = x    # y: seulement un alias de x,
         # pas de copie effectuee ici
y[2] = "tintin" # change x(2) en "tintin"
print(x) # [4, 2, "tintin", 9, "toto"]
x = [4, 2, 10, 9, "toto"]
y = x[:]   # On demande une copie
y[2] = "tintin" # modifie y mais pas x
print(x) # [4, 2, 10, 9, "toto"]
print(y) # [4, 2, "tintin", 9, "toto"]
```

# Conteneurs simples : tuple

**tuple** : Collection hétérogène, ordonnée et immuable

```
t=(5,7)
type(t) # tuple
t[0] # 5
t[0]=2 # error: item assignment for tuple
```

if - [elif] - [else]

```
if x < 0:
    print "x est negatif"
elif x % 2:
    print "x est positif et impair"
else:
    print "x n'est pas negatif et est pair"
```

# while

```
N = 0
x = input("Entrez un nombre positif x ")
while x > 0:
    x//=2
    N+=1
print("Approx. de log2(x): "+str(N-1))
```

for (et syntaxe des fonctions au passage)

```
for lettre in "ciao":  
    print lettre,
```

```
for x in ["\n",2,'a', 3.14,"\n"]:  
    print(x)
```

```
def f(n):  
    print(sum([k**2 for k in xrange(n)]))
```

```
def fib(n,a=0,b=1): #0,1: val. par default  
    '''n-th term of Fibonacci sequence...  
    starting at a,b.''' #tutoriel de fib  
    a, b = 0, 1  
    for i in xrange(n):  
        a,b=b,a+b  
    return a
```

La fonction doit être *chargée* (via une exécution) pour être appelée.



# Import de bibliothèques et de fonctions

Bibliothèques d'usage courant en calcul scientifique :

- **NumPy** : manipulation de tableaux numériques, fonctions mathématiques de base, simulation de variables aléatoires...
- **SciPy** : fonctions mathématiques plus avancées (résolution d'équations, d'équations différentielles, calcul d'intégrales...)
- **Matplotlib** : visualisation de données sous forme de graphiques
- **scikit-learn** : machine learning
- **SymPy** : calcul symbolique

Calcul numérique = manipulations de nombres décimaux  $\neq$  calcul symbolique = manipulation d'expressions symboliques

**Exemple** : racines de  $x^2 - x - 1 = 0$

$\hookrightarrow$  calcul symbolique :  $\frac{1+\sqrt{5}}{2}, \frac{1-\sqrt{5}}{2}$

$\hookrightarrow$  calcul numérique : 1.618034, - 0.6180340

# Import de bibliothèques ou de fonctions

```
import ma_bibliotheque  
# appel a une fonction de ma_bibliotheque:  
ma_bibliotheque.la_fonction(...)
```

```
import ma_bibliotheque as ma_bib #raccourci  
# appel a une fonction de ma_bibliotheque:  
ma_bib.la_fonction(...)
```

Moins précis (car la bibliothèque d'origine des fonctions n'est pas précisée à leur appel) :

```
from ma_bibliotheque import la_fonction  
# appel a une fonction de ma_bibliotheque:  
la_fonction(...)
```

```
from ma_bibliotheque import *  
# appel a la_fonction:  
la_fonction(...)
```

# Tableaux de nombres numpy

```
import numpy as np
b = np.array([[8,3,2,4],[5,1,6,0],[9,7,4,1]])
type(b) # numpy.ndarray
b.dtype # datatype: int
b.shape # (3,4)
c = np.array([[8,2],[5,6],[9,7]], dtype=complex)
c.dtype # datatype: complex
c[0,0] # 8+0j

#More than 2 dimensions:
d = np.array([[[8,3],[1,2]], [[5,1],[4,5]], [[9,7],[4,5]]])
d.shape # (3, 2, 2)
#Reshaping:
x = d.reshape(4,3) # tableau de taille (4,3)
d.reshape(12,1) # tableau de taille (12,1)
d.reshape(12,) # tableau unidimensionnel de taille 12
np.insert(np.arange(4,9),3,17) # 4,5,6,17,7,8
```

# Opérations sur les tableaux de nombres numpy

```
import numpy as np
X = np.arange(start=5, step=3, stop=16) # 5, 8, 11, 14
A = np.ones((2,3)) # matrix filled with ones
B = X.reshape(2,2)
C = np.zeros((3,2)) # matrix filled with zeros
D = np.eye(2) # identity matrix
np.diag([1,2]) # diagonal matrix
E = C+np.ones(C.shape) # addition: same as C+1
F = B*D # entry-wise multiplication
J=np.dot(B,D) # linear algebra product
G = F.T # transpose matrix
H = np.exp(G) #as most functions, exp is entry-wise
#(else use np.vectorize(my_function))
x = np.array([4, 2, 1, 5, 1, 10])
y=np.logical_and(x>=3, x<= 9, x!=1) # [T,F,F,T,F,F]
x[y] # [4, 5]
print(np.mean(np.random.randn(1e5)>1.96))
```

# Algèbre linéaire avec numpy

```
import numpy as np

A = [[2, 1, 1], [4, 3, 0]]
B = [[1, 2], [12, 0]]
C = [[1, 2], [12, 0], [-1, 2]]
D = [[1, 2, -4], [2, 0, 0], [1, 2, 3]]
E = np.bmat([[A, B], [C, D]]) # block matrix
type(E) # numpy.matrixlib.defmatrix.matrix

F = np.matrix(np.random.randn(5,5))
H = F*E # linear algebra product

B5 = np.linalg.matrix_power(B,5) # power
Bm1 = np.linalg.inv(B) # inverse
dB = np.linalg.det(B) # determinant
x = np.linalg.solve(B,[3,12])#solves B*x=[[3],[12]]
```

# Analyse spectrale avec numpy

```
import numpy as np

A = [[1, 2], [12, 3]]
x = np.linalg.eigvals(A) # eigenvalues
# eigenvalues and eigenvectors:
valp, vectp = np.linalg.eig(A)

#Hermitian matrices methods:
S = [[1, 2], [2, 3]]
y = np.linalg.eigvals(S) # eigenvalues
# eigenvalues and eigenvectors:
Valp, vVctp = np.linalg.eig(S)

#SVD:
U,s,V=np.linalg.svd(A)
Ap = np.matrix(U)*np.diag(s)*V
print(A-Ap)
```

## Copie de tableaux numpy

```
import numpy as np

x = np.array([[8,3,2],[5,1,0],[9,7,1]])
y = x
x[0,0]+=1
x[0,0]-y[0,0] # 0
z=x.copy()
x[0,0]+=1
x[0,0]-z[0,0] # 1
```

# Boucles vs programmation matricielle

```
import numpy as np
from time import time

n = int(1e7)
# Methode 1. Boucle for
t1 = time()
gamma1=sum([1./i for i in xrange(1,n)]) - np.log(n)
t2 = time()
temps1 = t2 - t1
# Methode 2. Numpy
t1 = time()
gamma2=np.sum(1. / np.arange(1,n)) - np.log(n)
t2 = time()
temps2 = t2 - t1
print "Facteur de gain: ", temps1/temps2
```

↪ éviter si possible boucles et tests en Python.



# Génération de variables aléatoires continues avec numpy

```
import numpy.random as npr  
npr.ma_loi(paramètres, taille_du_tableau)
```

- `npr.rand(d1,d2,...)` : tableau  $d1 \times d2 \times \dots$  de v.a.i. unif. sur  $[0, 1]$
- `npr.uniform(low=a,high=b,size=n)` : v.a.i. unif. sur  $[a, b[$  (`size=n` peut être remplacé par `size=(d1,d2,...)`, comme partout dans ce qui suit)
- `npr.randn(d1,d2,...)` : tableau  $d1 \times d2 \times \dots$  de v.a.i.  $\mathcal{N}(0, 1)$
- `npr.multivariate_normal(mean=V,cov=C,size=n)` : vecteurs aléatoires indépendants de loi  $\mathcal{N}(V, C)$  rangés dans un tableau de taille  $n \times N$ , où  $N$  est la taille de  $V$  et  $N \times N$  celle de  $C$
- `npr.exponential(scale=s,size=n)` : v.a.i. exponentielles de moyenne  $s$

Beaucoup d'autres exemples sur

<http://docs.scipy.org/doc/numpy/reference/routines.random.html>

# Génération de variables aléatoires discrètes avec numpy

```
import numpy.random as npr  
npr.ma_loi(paramètres, taille_du_tableau)
```

- `npr.rand(d1,d2,...)` : tableau  $d1 \times d2 \times \dots$  de v.a.i. unif. sur  $[0, 1]$
- `npr.randn(d1,d2,...)` : tableau  $d1 \times d2 \times \dots$  de v.a.i.  $\mathcal{N}(0, 1)$
- `npr.randint(low=a,high=b,size=n)` : v.a. unif. sur  $\llbracket a, b \llbracket$
- `npr.choice([a1,...,an],p=[p1,...,pn],size=n)` : tirages indép. dans  $[a1,...,an]$  de loi  $[p1,...,pn]$
- `npr.permutation(mon_urne)` : permutation de `mon_urne`
- `npr.binomial(n,p,size=n)`
- `npr.geometric(p,size=n)`
- `npr.multinomial(n,tableau_des_probab,size=n)`
- `npr.poisson(alpha,size=n)`

Beaucoup d'autres exemples sur

<http://docs.scipy.org/doc/numpy/reference/routines.random.html>

# Fonctions utiles en probabilités

- **np.mean(x)**, **np.std(x)**, **np.percentile(x)** :  
moyenne, écart-type et percentile d'un vecteur **x** (échantillon)
- **np.sum(x)** somme des valeurs de **x**
- **np.cumsum(x)** vecteur  $[x_1, x_1 + x_2, \dots, x_1 + \dots + x_n]$  des  
sommes cumulées des coordonnées  $x_1, \dots, x_n$  de **x**
- **np.cov(x)** matrice  $n \times n$  covariance des lignes du tableau **x**  
de taille  $n \times p$
- **scipy.stats** : bibliothèque proposant **densités**, **fonctions  
de répartition**, **quantiles**, etc... de lois classiques. Cf  
<http://docs.scipy.org/doc/scipy/reference/stats.html>
- **matplotlib.pyplot** bibliothèque d'affichage graphique

# Affichage graphique avec matplotlib.pylab

```
import matplotlib.pylab as plt
```

Pour **x**, **y** vecteurs de même dimension,

- **plt.plot(x,y)** affiche la courbe affine par morceaux reliant les points d'abscisses **x** et d'ordonnées **y** (nombreuses options)
- **plt.hist** trace un histogramme (spécifier **normed=True**). Deux options pour les colonnes : **bins= nombre de colonnes** ou **bins= abscisses des séparations des colonnes**
- **plt.bar** trace un diagramme en bâton
- **plt.scatter(x,y)** affiche le nuage de points d'abs. **x** et d'ord. **y**
- **plt.stem(x,y)** affiche des barres verticales d'abs. **x** et hauteur **y**
- **plt.axis([xmin,xmax,ymin,ymax])** définit les intervalles couverts par la figure, alors que **plt.axis('scaled')** impose que l'échelle en **x** et l'échelle en **y** soient les mêmes

# Affichage graphique avec matplotlib.pyplot

```
import matplotlib.pyplot as plt
```

- `plt.show()` affiche les fenêtres créées dans le script
- `plt.figure()` crée une nouvelle fenêtre graphique
- `plt.title("mon titre")` donne un titre à une figure
- `plt.legend(loc='best')` affiche la légende d'un graphique (en position optimale)
- `plt.subplot` subdivise la fenêtre graphique de façon à y afficher plusieurs graphiques dans la même figure

# Histogramme d'un échantillon de loi continue

```
import numpy as np
import scipy.stats as sps
import matplotlib.pyplot as plt

E = np.random.randn(1e5) #echantillon

x = np.linspace(-4,4,1000)
f_x = sps.norm.pdf(x) #Densite gaussienne
plt.plot(x,f_x,"r")
#Affichage histo:
plt.hist(E,bins=50,normed=1,histtype='step')
plt.legend(loc=4)
plt.show()
```

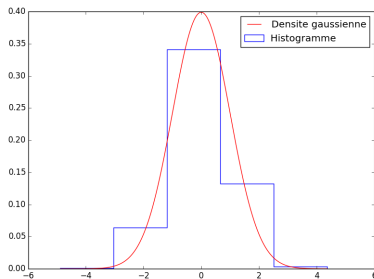
Cf. [RepLoiContinue.py](#) pour un programme plus détaillé.

# Représentation d'un échantillon de loi discrète

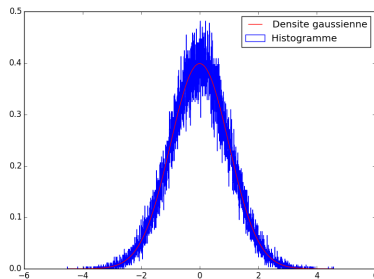
```
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt
plt.close("all")

n, p, N = 20, 0.3, int(1e4)
B = np.random.binomial(n, p, N)
f = sp.stats.binom.pmf(np.arange(n+1), n, p)
plt.hist(B, bins=n+1, normed=1, range=(-.5, n+.5), \
         color="white", label="loi empirique")
plt.stem(np.arange(n+1), f, "r", label="loi theorique")
plt.legend()
plt.title("Binomiale vs Poisson")
plt.show()
```

# Représentation d'un échantillon de loi continue : combien de colonnes dans l'histogramme ?



(a) Trop peu de colonnes



(b) Trop de colonnes

Idéal  $\approx N^{1/3}$  colonnes ( $N$  : taille de l'échantillon)

Cf. [SliderHistograms.py](#) (à exécuter dans une console [Python](#) car dynamique) pour une approche empirique.



## Théorème (*Loi des Grands Nombres*, Kolmogorov, 1929)

$(X_i)_{i \geq 1}$  v.a.i.i.d. :

$$\begin{aligned}\mathbb{E}[|X_1|] < \infty &\implies \frac{X_1 + \dots + X_n}{n} \xrightarrow[n \rightarrow \infty]{p.s.} \mathbb{E}[X_1], \\ \mathbb{E}[|X_1|] = \infty &\implies \frac{X_1 + \dots + X_n}{n} \text{ diverge p.s..}\end{aligned}$$

## Théorème (*Central Limite*, Laplace, 1812, Lindeberg, 1920)

$(X_i)_{i \geq 1}$  v.a.i.i.d. d'espérance  $\mu$  et d'écart-type  $\sigma$  fini :

$$n^{1/2} \left( \frac{X_1 + \dots + X_n}{n} - \mu \right) / \sigma \xrightarrow[n \rightarrow \infty]{loi} \mathcal{N}(0, 1).$$

*Moyenne empirique des  $X_i \approx$  moyenne théorique, avec  
une erreur gaussienne d'ordre  $1/\sqrt{n}$*

*Moyenne empirique des  $X_i \approx$  moyenne théorique (LGN),  
avec une erreur **gaussienne** d'ordre  $1/\sqrt{n}$  (TCL) si  
**espérance finie** (LGN) et **écart-type fini** (TCL)*

Soit  $s, U$  v.a. indépendantes,  $U$  uniforme sur  $[0, 1]$  et  $s = \pm 1$  avec probas 0.5, 0.5. Alors la v.a.  $X := sU^{-1/\alpha}$ , appelée ici  **$\alpha$ -variable aléatoire**, a pour densité  $(\alpha/2)\mathbb{1}_{|x|\geq 1}|x|^{-\alpha-1}$

$\hookrightarrow$  **espérance finie** si  $\alpha > 1$  et **écart-type fini** si  $\alpha > 2$ .

Les scripts **SliderLGN.py** et **SliderTCL.py** (à exécuter dans une console **Python** car dynamiques) testent ces théorèmes pour les  **$\alpha$ -variables aléatoires** pour différentes valeurs de  $n$  et de  $\alpha$  :

- **SliderLGN.py** : convergence ou non de la moyenne empirique  $\frac{X_1 + \dots + X_n}{n}$  (transition à  $\alpha = 1$ )
- **SliderTCL.py** : histogramme de l'erreur en fonction de  $\alpha$  (transition théorique à  $\alpha = 2$ )

# Lecture et écriture dans un fichier externe

```
x=range(5)
#Creation et ecriture:
mon_flux=open("my_data.txt","w") #w=write
mon_flux.write(str(x))
mon_flux.close()

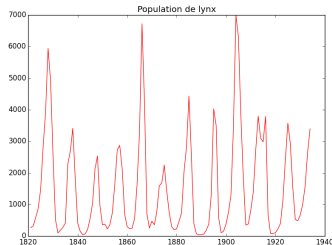
#Ecriture a la fin d'un fichier existant:
mon_flux=open("my_data.txt","a") #a=append
mon_flux.write("\n"+str(x+1))
mon_flux.close()

#Lecture:
mon_flux=open("my_data.txt","r") #r=read
y=mon_flux.read()
print(y)
```

Voir aussi `np.save`, `np.load`, `np.savetxt`...

# Lecture dans un fichier externe : exemple

```
import numpy as np
import matplotlib.pyplot as plt
plt.close("all")
f=open("PopLynxRegionCanada_1821_1934.dat","r")
ytxt=f.readlines() # list de str
y=[int(row) for row in ytxt] # convertit str en int
plt.plot(range(1821,1935),y,"r")
plt.title("Population de lynx")
plt.tight_layout() #pratique pour l'export
plt.show()
```



# Python pour l'analyse : exemples

```
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt
def f(x):
    return np.exp(x)+x
#zeros of f, computed starting at -0.2:
a=sp.optimize.fsolve(f,-.2)
print(f(a))
#integral of f from 0 to 1:
b=sp.integrate.quad(f,0,1)
def g(y,t):
    return y
T=np.arange(0,100)/100.
#solution, at T, of  $y'=g(y,t)$  starting at 1 at T[0]:
y=sp.integrate.odeint(g,1,T)
plt.plot(T,np.log(y),"r")
plt.show()
```

# Bibliographie : les tutoriels/sites officiels

- **Python** : <https://docs.python.org/2/tutorial/>
- **NumPy** : <http://docs.scipy.org/doc/numpy/reference/>
- **ScipyStats** : <http://docs.scipy.org/doc/scipy/reference/tutorial/stats.html>
- **Matplotlib** :  
[http://matplotlib.org/users/pyplot\\_tutorial.html](http://matplotlib.org/users/pyplot_tutorial.html)
- **NumPy user guide (pdf)** : <https://docs.scipy.org/doc/numpy-1.8.0/numpy-user-1.8.0.pdf>
- **Matplotlib user guide (pdf)** :  
<http://matplotlib.org/Matplotlib.pdf>
- **scikit-learn** : <http://scikit-learn.org/stable/>
- **SymPy** : <http://www.sympy.org/fr/index.html>
- **Anaconda (distribution contenant l'interface de développement Spyder)** : <https://www.continuum.io/downloads>

# Bibliographie : quelques cours

- **Un cours de l'X** : [http://www.cmap.polytechnique.fr/~gaiffas/intro\\_python.html](http://www.cmap.polytechnique.fr/~gaiffas/intro_python.html)
- **Un très bon cours du lycée Saint Louis** :  
<http://mathprepa.fr/python-project-euler-mpsi/>
- **Un cours de l'INRIA** : <http://www.labri.fr/perso/nrougier/teaching/index.html>
- **Un cours d'Orsay** : [http://www.iut-orsay.u-psud.fr/fr/specialites/mesures\\_physiques/mphy\\_pedagogie.html](http://www.iut-orsay.u-psud.fr/fr/specialites/mesures_physiques/mphy_pedagogie.html)